Implementation of BT, SP, LU, and FT of NAS Parallel Benchmarks in Java

Matthew Schultz, Michael Frumkin, Haoqiang Jin, and Jerry Yan[*]
Numerical Aerospace Simulation Systems Division
NASA Ames Research Center

Abstract

A number of Java features make it an attractive but a debatable choice for High Performance Computing. We have implemented benchmarks working on single structured grid BT,SP,LU and FT in Java. The performance and scalability of the Java code shows that a significant improvement in Java compiler technology and in Java thread implementation are necessary for Java to compete with Fortran in HPC applications.

1. Introduction

The portability and expressive power of Java language ignited interest in the HPC community to use Java for computationally intensive tasks. While there are number of significant obstacles to achieve high performance of Java code there is work under way to implement a highly efficient Java compiler [5]. In this paper we evaluate Java as a choice for programming of Aerospace applications. For this purpose we have implemented NAS Serial Benchmarks BT, SP, LU, and FT in Java, parallelized them using Java threads and collected some performance data on SUN Enterprise10000 and Origin2000 machines. The serial code used in the development is the same code used in HPF versions of the benchmarks [2].

The implementation includes three stages: translation, multithreading and performance analysis. The translation phase consisted of developing a strategy for mapping Fortran constructs to Java constructs and creating macros to automate the process. The parallelization phase consisted of creating a base class to hold the data to be shared among threads, deriving a new class for each type of thread and writing code to split the work between the threads. The performance analysis stage includes profiling code, comparing the performance with the Fortran version and analyzing Java overhead and threading overhead.

[*]MRJ Technology Solutions, Inc. M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000; e-mail: {frumkin,yan}@nas.nasa.gov

2. Fortran to Java Translation Options

Java is more expressive language than Fortran. It implies a simple translation from Fortran to Java and a challenge in achieving performance of Java code matching the performance of Fortran code. In the process of translating the NAS Parallel Benchmarks [1] written in Fortran to Java we have to address a number of issues: conversion of arrays, DO loops, conditional statements and numerical stability.

In order to convert Fortran to Java, a consistent method for converting arrays had to be developed. The issue is simplified by the fact that both Fortran and Java employ a linear memory model. Fortran arrays are simply pointers to contiguous blocks of memory while Java arrays are objects. The consequence of this is that in Fortran, arrays can start from any index whereas in Java the index must always be zero. The default starting index of a Fortran array is one. This means that to directly convert the default Fortran array to Java, an array subscript expression must be shifted by one for each dimension.

In Java, multidimensional arrays are implemented as arrays of arrays, and because of the promise of Java to be a safe language, a bounds check is required for each dimension of the array.

In Fortran, a subscripted array is passed to a function as a pointer to, that location in the array. Both these factors make it desirable to implement Fortran arrays in Java as single dimensional arrays. This reduces the bounds checking to two comparisons as well as allowing Fortran functions taking arrays to be passed with only one additional argument for the offset to the array. Multidimensional arrays in Java we implemented as one dimensional arrays using index linearization to simulate multiple dimensions.

Example 1. Fortran array

```
REAL*8 u(5,nx,ny,nz)
u(m,i,j,k)=...
```

we translate to Java code:

```
double u[]=new double[5*nx*ny*nz];
int usize1=1,
    usize2=usize1*5,
    usize3=usize2*nx,
    usize4=usize3*ny;
u((m-1)*usize1+(i-1)*usize2+(j-1)*usize3+(k-1)*usize4)=...
```

The Fortran dimension statement can be directly translated to the resetting array sizes, however in the benchmarks it was required only few times and we did it by hand.

The described mapping has one deficiency making programming with one dimensional Java arrays difficult. The problem is that usize* should be recomputed each time array u is redimensioned, for example in a subroutine. This programming hardship can be alleviated by simulating Fortran arrays with a ShapedArray class:

```
class ShapedArray{
  private double el[];
  private int size1,size2,size3;
  private void shape(int k,int l,int m){
    size1=k; size2=l; size3=m;
  }
  private int idx(int i,int j,int k){
    int offset=(i-1)*usize1+(j-1)*usize2+(k-1)*usize3;
    if(offset<0||offset>=el.length)
      throw(InvalidArrayIndexException);
    return offset;
  }
};
```
Array access would look as follows:

```
ShapedArray u = new ShapedArray[63];
/*reshaping the array*/
u.shape(7,3,3);
/*accessing array element based on the current shape*/
u.el[u.idx(i,j,k)]=...
```

Use of the ShapedArray would make Java code simpler to read and more resembling Fortran code, however it would involve a call to a member function at each array access. Other array implementations in Java such as Java Array Package [6] would suffer the same inefficiency problem unless the Java compiler supports semantic inlining.

Java lacks primitive complex data type used in FT benchmark. The use of Complex class would cause the similar inefficiency as the use of Array class. For this reason we have implemented complex arrays as array of doubles of twice of size with even elements storing real parts and odd elements storing the imaginary parts. The modification of the FT code performing actual calculations with complex numbers was done by hand.

3. Semiautomatic Fortran to Java Translation

For converting Fortran array notation to Java notation, we used emacs regular expressions (RE) to convert the vast majority of the array expressions. In Fortran, an array is an identifier followed by an open parenthesis, followed by a comma separated list of expressions, followed by a closing parenthesis. In emacs, the value of an RE can be saved by enclosing it in backslashed parentheses and later referenced by using backslash and the number, in order, of the saved RE. These factors make it possible to translate arrays mechanically on a per array basis using the following macro.

```
arrayname(<[^,]+>,<[^,]+>,<[^,]+>,<[^)]+>)  =>
arrayname[\1+\2*size1+\3*size2+\4*size3]
```

For an array with n dimensions the first n-1 array expressions can be read in by first finding `arrayname(`, and then getting one or more characters that are not a comma `<[^,]+>` for each dimension up to n-1. The last dimension is read in by getting one or more characters that are not a closing parenthesis `<[^)]+>`. This is then replaced with

```
arrayname[\1+\2*size1+\3*size2+\4*size3]
```

The final subscript decrementing was done by hand.

The basic do loop with an increment of one have been converted to Java for loops using the macro

```
do[ ]+<[-+a-z0-9]+>[ ]*=[ ]*<[-+a-z0-9]+>[ ]*,[ ]*<.+>
```

and replacing it with

```
for(\1=\2;\1<=\3;\1++){
```

The `end do` can then have been converted to a closing brace using the macro `end[ ]+do`.

Translation of general Fortran conditional statements to Java would require a parsing of Fortran code and translation of the parse tree to Java equivalent. However, the translation of simple conditional `if then else` chains found in the benchmarks have been accomplished in 4 steps:

- transform all `then` to `{`,
- transform all `else if` to `}else if`,
- transform all `else` to `}else{`
- transform `endif` to `}`.

Several Fortran constructs can be changed to Java through context free replacement. These include all boolean operators, all type declarations except character arrays which should be converted to Java strings, comments, the call statement, and line continuation marker.

Using these macros as well as hand coding end of statements, intrinsic functions, array bounds and various miscellaneous Fortran code the BT, SP, LU, and FT have been translated to Java in the following way. We created a class for each benchmark and made function skeletons for each Fortran function in each benchmark. We then made all common variables members of the benchmark class. We also created stride variables for each different size of array. We then created a function skeleton for the main program called `runBenchMark`. Then we inserted the Fortran code for each function and translated it directly into Java.

4. Using Java Threads for Parallelization

Conceptually Java Threads are close to the OpenMP threads and we used OpenMP version of the benchmarks (PBN-OMP), implemented in NASA Ames Research Center, as a starting point for writing parallel versions. The base class for threads was derived from class `java.lang.Thread` and all common variables were implemented as static members of this class. Then we derived from this base class, classes for each parallelizable routine and a master class to control the synchronization of the worker classes. To implement synchronization by switching threads between blocked and runnable states we used the `wait()` and `notify()` methods of the `Thread` class and a boolean variable to denote when a worker was done with a given task.

The work of initializing and partitioning the work for the threads is accomplished in the benchmark base constructor along with setting up the benchmark for a given size class. Two functions, `partition` and `set_interval` are used to split the outer `for` loops of parallelizable routines among the threads. The `partition`, takes a given size and number of threads as arguments and splits the number of loop iterations as evenly as possible among the threads. The `set_interval` then takes an array of intervals and a starting point and returns a starting and ending point for each threads' `for` loop.

The master thread dispatches the job to each worker, starts workers and waits until all workers are finished:

```
for(i=0;i<num_threads;i++)worker[i].done=false;
for(i=0;i<num_threads;i++)
  synchronized(worker[i]){worker[i].notify();}
for(i=0;i<num_threads;i++)
  while(!worker[i].done){
        try{wait();}catch(Exception e){}
    }
```

Each worker thread is then started and immediately goes into a blocked state on the condition that the variable done is true, then performs the work and notifies the master that the work is done:

```
public synchronized void run(){
   while(true){
      while(done){
        try{wait();} catch(InterruptedException ie){}
      }
      step();
      done=true;
      synchronized(master){master.notify();}
   }
};
```

All worker threads are implemented as daemon threads, so that when the master thread terminates, the program will automatically terminate even though the worker threads are never explicitly destroyed. The while loop around the wait call prevents an arbitrary notify call from waking a thread before its time. All CFD code is placed in the step method. When the step method returns, the thread sets it's done flag to true and then notifies the master thread that it is done.

The described model of thread synchronization is applicable only if there is no dependence between workers: each worker processes the job dispatched by the master independently on other workers. Such dependence, however exists in LU where the computations are pipelined. This requires for each worker to notify its neighbor on the work ready for his execution. We have implemented the pipelined computations with two level wait-notify methods in worker threads:

```
while(true){
  while(done){
    try{wait();}catch(InterruptedException ie){}
  }
  for(k=1;k<nz;k++){
    if(id>0){
      while(todo<=0){
        try{wait();}catch(Exception e){}
      }
    }
    step(k);
```

```
        todo--;
        if(id<num_threads-1)synchronized(neighbor[id+1]){
          worker[id+1].todo++;
          worker[id+1].notify();
        }
      }
      done=true;
      if(id==num_threads-1)
        synchronized(master){master.notify();}
    }
```

The master thread starts all workers but waits only for the last worker to finish (the pipeline mechanism guarantees that all other workers have finished already):

```
    for(i=0;i<num_threads;i++) worker[i].done=false;
    for(i=0;i<num_threads;i++)
      synchronized(worker[i]){worker[i].notify();}
    while(!worker[num_threads-1].done){
      try{wait();} catch(Exception e){}
    }
```

5. The Performance and Speedup

The performance of a Java implementation of the NAS Parallel Benchmarks is significantly slower than the corresponding Fortran code. Because the serial benchmarks were directly translated, the slowdown must be attributed to the JVM. Several factors are involved in this slowdown.

Although the majority of bounds checks were eliminated through flattened arrays, one pair of bounds checks for each array access is still done. Coupled with the fact that the majority of the array accesses are within for loops, the bounds checking done by the JVM slows down the NPB codes. In [5] it is shown that moving bounds check out of the loop nest can reduce the number of bound checks to few checks per loop nest, while preserving the Java requirements for secure array access. This reduction, however, requires some support from JVM.

Another slowdown occurs because of the virtual function mechanism in Java, which requires an additional pointer dereference. As the statistics below show, the factor by which the benchmarks slow down is machine and JVM dependent. All benchmarks have bee compiled with flag -O and on SGI machine have been run with -native flag to JIT.

A comparison of Fortran code and Java code running under `perfex` on Origin2000 showed that the Java code has an order of magnitude more instructions than similar fortran code. The perfex profile also shows that Java code has better cache utilization then the Fortran code which we attribute to the spilling of instructions to the cache. The number of floating point instructions issued by Java code was about factor of 2 larger then in the Fortran code, confirming known Java inability to issue the mad instruction, cf. [5].

Performance of the Java code is summarized in Table 1, Table 2, and the profile of the subroutine performance is shown in Figure 1, Figure 2, Figure 3.

TABLE 1. Benchmarks time on SUN Enterprise10000 (333 MHzmachine)(sec), Java version "1.1.3"

Thread number	Serial	1	2	4	8	9	12	16
BT.A Java	13609.5	14671.3	7381.7	3846.3	2305.0	2042.7	1782.7	1762.2
SP.A Java	10235.8	11108.1	5692.9	3409.3	2095.5	1899.1	1862.1	1671.2
LU.A Java	12344.5	13578.9	6843.3	3765.7	2077.3	1892.7	1730.2	1745.4
FT.A Java	1104.6							

TABLE 2. Benchmarks time on SGI Origin2000 (195 MHz machine) (sec), Java version "3.1 (Sun 1.1.5)"

Thread number	Serial	1	2	4	8	9	12	16
BT.A Java	14605.2	16447.6	8448.4	4375.6	2550.9	2230 2	1700.6	1367.4
SP.A Java	9948.0	10293.7	5640.4	3001.4	1699.3	1690.3	1361.3	1254.9
LU.A Java	14684.8	17585.7	10330.0	4901.6	2511.1	2335.2	1797.7	1655.9
FT.A Java								

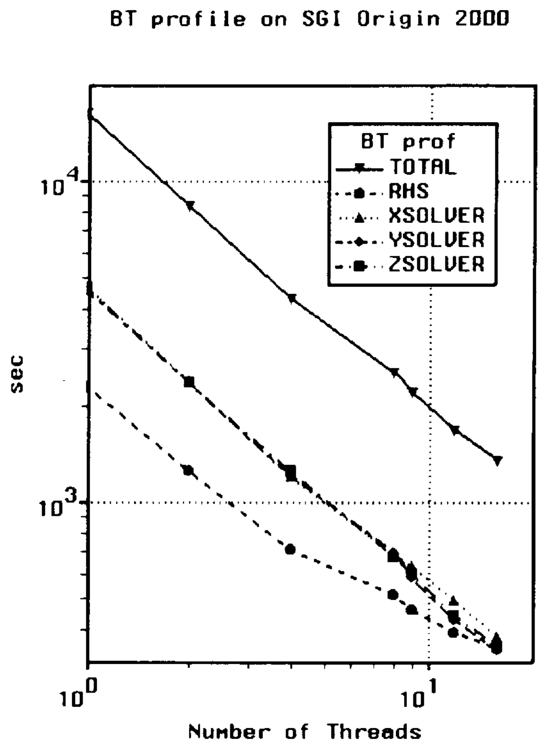
BT profile on SGI Origin 2000
BT prof
TOTAL
RHS
XSOLVER
YSOLVER
ZSOLVER
sec
10^4
10^3
10^0
10^1
Number of Threads

FIGURE 1. BT profile on Origin2000

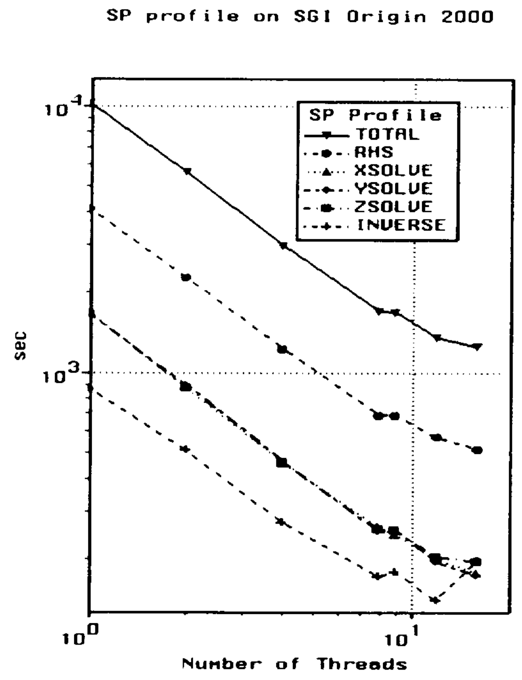

FIGURE 2. SP profile on Origin2000

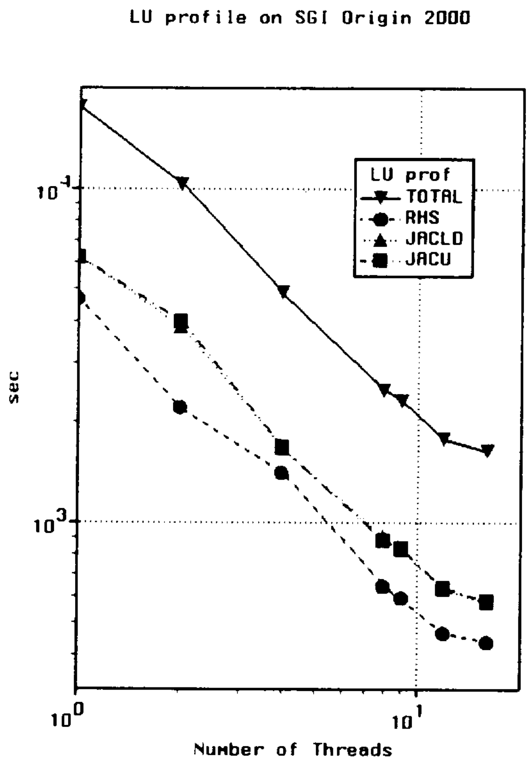

FIGURE 3. LU profile on Origin2000

Multithreading of the Java Benchmarks introduces overhead about 10%. The speed-up with 16 threads is in the range 6-12 (efficiency 0.38-0.75).

6. Related Work

In implementing BT, SP and LU, only Java threads were used. The University of Westminster's Performance Engineering Group at the School of Computer Science used the Java jni to create a system dependent Java MPI library. They also used this library to implement the NAS benchmarks FT and IS using javaMPI [3].The Westminster version of javaMPI can be compiled on any system with java 1.0.2 and LAM 6.1.

The University of Adelaide's Distributed and High Performance Computing Group, has also released the NAS benchmarks EP and IS (with FT, CG and MG under development), [4] along with many other benchmarks in order to test Java's suitability for grand applications.

7. Conclusions

Although the performance of these benchmarks is not comparable with other languages at this time, using the performance enhancing methods detailed in [5] serial performance could be improved to near Fortran-like performance. Efficiency of paralleization with threads is about 0.5 fo up to 16 threads and comparable with efficiency of parallelization with OpenMP, MPI and HPF. However, with several groups working on MPI for Java, improvements in parallel performance and scalability seem inevitable as well. The attraction of Java as a numerically intensive applications language is primarily driven by its ease of use, portability and widespread documentation and reference materials. If it is made to run faster though methods that have already been researched extensively, such as high order loop transformations, semantic inlining and a wider availability of traditionally optimized native compilers, it could be an ideal environment for such applications.

8. References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* Report NAS-95-020, Dec. 1995. http://science.nas.nasa.gov/Software/NPB.

[2] M. Frumkin, H. Jin, J. Yan. *Implementation of NAS Parallel Benchmarks in High Performance Fortran.* CDROM version of IPPS/SPDP 1999 Proceedings, April 12-16, 1999, San Juan, Puerto Rico, 10 pp.

[3] V.Getov, S. Flynn-Hummel, S. Mintchev. *High-Performance Parallel Programming in Java: Exploiting Native Libraries.* Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing, 10 pp., http://perun.hscs.wmin.ac.uk/JavaMPI.

[4] J.A.Mathew, P.D.Coddington and K.A.Hawick. *Analysis and Development of Java Grande Benchmarks.* Proc. of the ACM 1999 Java Grande Conference, San Francisco, June 1999, 14 pp., http://www.cs.adelaide.edu.au/users/paulc.

[5] S.P. Midkiff, J.E. Moreira, M Snir. *Java for Numerically Intensive Computing: from Flops to Gigaflops.* Proceedings of FRONTIERS'99, pp. 251-257, Annapolis Maryland, February 21-25, 1999.

[6] S.P. Midkiff, J.E. Moreira, M. Gupta, F. Artigas, *Numerically Intensive Java.* http://www.alphaWorks.ibm.com/tech/ninja.